Program Synthesis from Natural Language Using Recurrent Neural Networks

Presented by Dewi Yokelson, April 2019

Paper by Xi Victoria Lin & Others
• Make programming easier and more productive by letting programmers use their own words and concepts to express the intended operation
• Avoid wasted time searching online when the programmer does not know key words to search or cannot find the answer
• Man pages can be hard to discover and understand
Question 1. I have a bunch of “.zip” files in several directories “dir1/dir2”, “dir3”, “dir4/dir5”. How would I move them all to a common base folder? (http://unix.stackexchange.com/questions/67503)

Solution: find dir*/ -type f -name "*.zip" -exec mv {} "basedir" \;

Question 2. I have one folder for log with 7 sub-folders. I want to delete all the files older than 15 days in all folders including sub-folders without touching folder structure. (http://unix.stackexchange.com/questions/155184)

Solution: find . -type f -mtime +15 | xargs rm -f
• Does the translation using recurrent neural networks (RNNs)
• An interactive web page where you type in your natural language statement and you receive a ranked list of possible bash one line commands
• http://tellina.rocks
Recurrent Neural Networks

- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

- Traditional neural nets accept fixed sized vectors as input and produce fixed sized output, not so with RNNs
The Approach

- User provides natural language sentence X which Tellina transforms into a template
- An RNN encoder-decoder model translates the template into a ranked list of possible program templates with argument slots
- The argument slots are replaced by program literals to produce an output program using a k-nearest neighbor classifier
Template Generation

- Used a domain-specific heuristic: defined two categories of entities, *patterns* and *quantities*
- To recognize and assign types to natural language commands they manually defined regexs and mapped them to their type
- To recognize and assign types in the bash command templates they map man page types to the types above
Algorithm 1: Global entity-slot alignment

**Input**: List of entities $E$, list of argument slots $S$, local entity-slot compatibility function $\gamma(i,j)$.

**Output**: List of matched entity-slot pairs $M$ if every entity is aligned to a slot; null otherwise.

1. $M = \emptyset$
2. /* compute the preference list for each entity */
   
   for $e_i \in E$ do
     
     PriorityQueue $S_{e_i}$;
     
     for $s_j \in S$ do
       if $\gamma(i,j) \neq \text{inf}$ then
         $S_{e_i}$.Enqueue($s_j$)
       end
     end
   end
3. /* compute the stable alignment */
   
   while $\exists e_i$ s.t. $\forall s_j (e_i, s_j) \notin M \land S_{e_i} \neq \emptyset$ do
     $s_j = S_{e_i}$.Dequeue();
     
     if $\exists e_i'$ s.t. $(e_i', s_j) \in M$ then
       if $\gamma(i', j) < \gamma(i, j)$ then
         $M = M \cup \{(e_i, s_j)\} \setminus \{(e_i', s_j)\}$;
       end
     else
       $M = M \cup \{(e_i, s_j)\}$;
     end

end

• Often a one to one mapping between the entities in NL and the resulting program

• Tellina aligns the most likely entities using the Global Entity-Slot Alignment algorithm (previous slide) and then extracts the values from the NL sentence and inserts them into the program

• Each entity-slot pair \((e_i, s_j)\) is represented using the concatenation of the hidden state vectors \((h_i, h'_j)\) of the neural encoder-decoder model
• Labor-intensive data collection process: hired workers to scrape the web for ultimately just over 5000 nl-bash pairs

<table>
<thead>
<tr>
<th>In-scope syntax structures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Single command</td>
</tr>
<tr>
<td>• Logical connectives: &amp;&amp;,</td>
</tr>
<tr>
<td>• Nested commands: pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out-of-scope syntax structures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O redirection &lt;, &lt;&lt;</td>
</tr>
<tr>
<td>• Variable assignment =</td>
</tr>
<tr>
<td>• Parameters $1</td>
</tr>
<tr>
<td>• Compound statements: if, for, while, until, blocks, function definition</td>
</tr>
<tr>
<td>• Regex structure (every string is a single opaque token)</td>
</tr>
<tr>
<td>• Non-bash program strings triggered by command interpreters such as awk, sed, python, java</td>
</tr>
</tbody>
</table>
### Table 2: Translation accuracies of the Tellina model and the code retrieval baseline.

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc$_F^1$</th>
<th>Acc$_F^3$</th>
<th>Acc$_T^1$</th>
<th>Acc$_T^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR Baseline</td>
<td>13.0%</td>
<td>20.6%</td>
<td>54.7%</td>
<td>67.9%</td>
</tr>
<tr>
<td>Tellina Model</td>
<td>30.0%</td>
<td>36.0%</td>
<td>69.4%</td>
<td>80.0%</td>
</tr>
</tbody>
</table>

### Table 3: Development set performance of the argument filling component for differing $k$ nearest neighbor values.

<table>
<thead>
<tr>
<th>$k$</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82.9</td>
<td>87.0</td>
<td>84.9</td>
</tr>
<tr>
<td>5</td>
<td>84.6</td>
<td>89.0</td>
<td>86.7</td>
</tr>
<tr>
<td>10</td>
<td>82.1</td>
<td>86.2</td>
<td>84.1</td>
</tr>
<tr>
<td>100</td>
<td>79.8</td>
<td>84.0</td>
<td>81.9</td>
</tr>
<tr>
<td>200</td>
<td>77.2</td>
<td>81.2</td>
<td>79.1</td>
</tr>
</tbody>
</table>
• Conducted a user study to determine whether Tellina helps programmers complete file system tasks using bash
• Recruited 39 CS students, all familiar with bash
• Assigned 2 tasksets made up of 8 tasks, for each taskset they were either allowed to use Tellina or not
• Overall success rate 88%, participants using Tellina on average used 22% less time and had a 90% success rate over the 85% in the control group (without Tellina)
Questions?